Khoo Wei Ping - Project Portfolio

Project: Codii

[Codii (https://github.com/CS2103AUG2017-T17-B1/main)] is a desktop address book application specially designed for debt collectors to manage debtors in a simple manner. It has a GUI but most of the user interactions happen using a CLI (Command Line Interface).

Debt collectors can store information such as the amount owed, debt borrow date and debt cleared date in addition to debtor's personal information.

Unique features such as an interest calculator help debt collectors manage debts more efficiently.

Codii is evolved from AddressBook - Level 4 (https://github.com/nus-cs2103-AY1718S1/addressbook-level4), which is a desktop address book application used for teaching Software Engineering principles.

Code contributed: [Functional code (https://github.com/CS2103AUG2017-T17-B1/main/tree/master/collated/main/khooroko.md)] [Test code (https://github.com/CS2103AUG2017-T17-B1/main/tree/master/collated/test/khooroko.md)]

Enhancement Added: Storage backup

External behavior

Start of Extract [from: User Guide]

Saving the data

Address book data is saved in the hard disk automatically after any command that changes the data.
There is no need to save manually.
If address book data can be loaded successfully, backup address book data is saved upon starting the program.

Loading the data

If the data file does not exist or cannot be read:

Backup data file will be loaded, if available and readable.

If backup data is unavailable:

You will be given a sample address book.

If backup data exists but cannot be read :

You will be given an empty address book.

To quickly revert address book data to the state of last use:

1. Delete addressbook.xml.
2. Rename addressbook.xml-backup.xml to addressbook.xml.

End of Extract

Justification

The backup is a safety measure in case the main storage file is corrupted or deleted by accident.

Implementation

Start of Extract [from: Developer Guide]

Backup storage mechanism

The backup storage mechanism is facilitated by the `StorageManager`. It backs up the address book data automatically each time the application starts up, if there is existing data available. The sequence diagram for this is shown below in Figure 4.3.

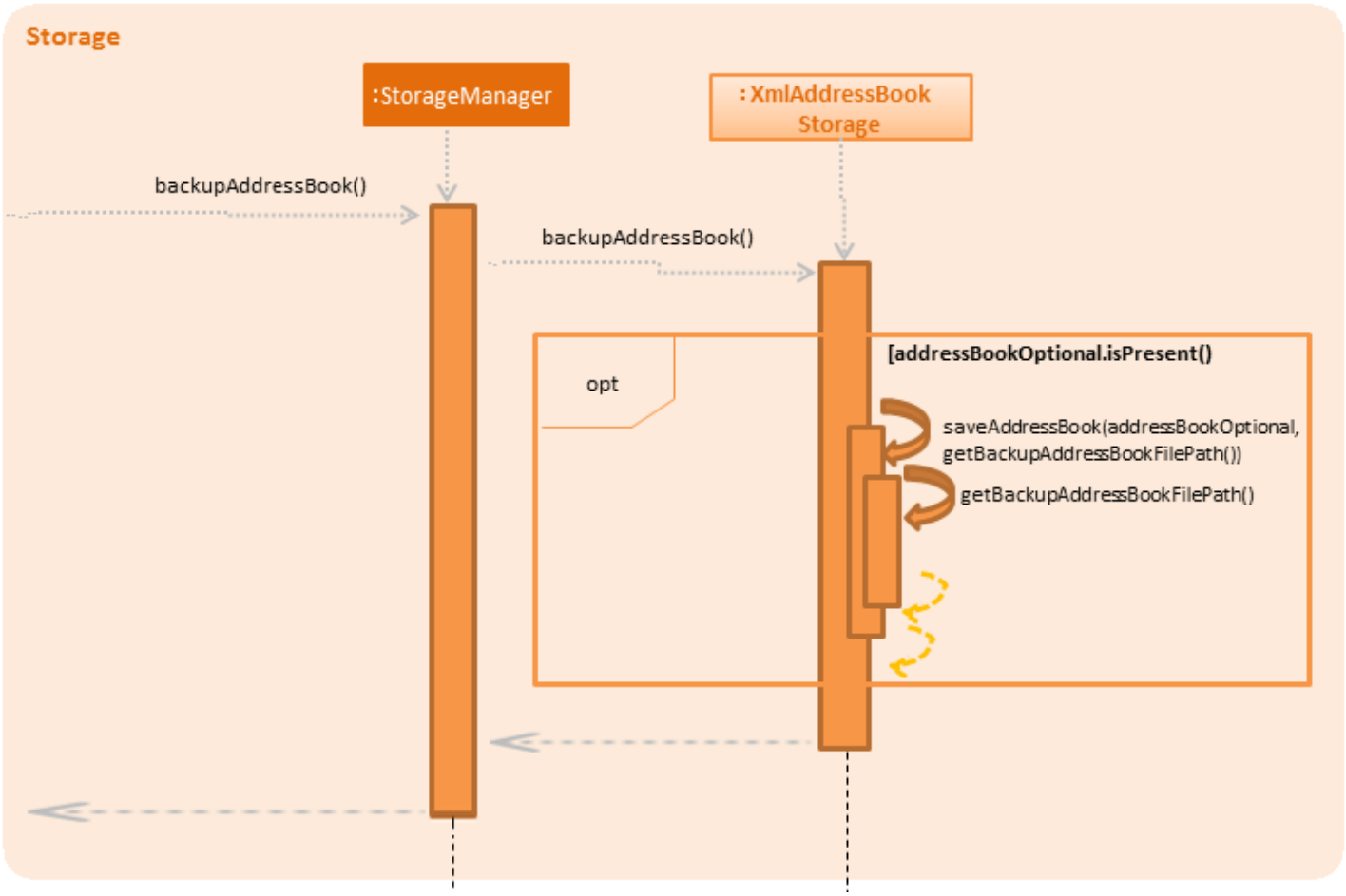

Figure 4.3.1: Sequence diagram for backing up address book data

The `backupAddressBook` method is called in `MainApp#init()` which is called each time the application starts. No backup is made if there is no existing data.

Design Considerations

Aspect: When to create the backup
Alternative 1 (current choice): Create it on application start up
Pros: Guarantees that a backup is made only of a working version of the address book that the user can easily revert to, should he/she mess up the main copy while using the application.
Cons: Not suitable for users who tend to make many changes within one session, as too many changes will not be backed up.
Alternative 2: Implement a command to create backup
Pros: The backup is only created when the user desires to.
Cons: This makes the implementation pointless altogether as it is meant as a safety net for clumsy users. This alternative would only benefit the careful users, who do not require it as much in the first place.
Alternative 3: Create a backup after a fixed number of commands that change the address book
Pros: This will create backups that are guaranteed to be recent.
Cons: It is difficult to determine the ideal number of commands to ensure that the backup is both recent enough, yet outdated enough for the user to want to restore state to should he/she mess up a command at some point.

End of Extract

Enhancement Added: Sort command

External behavior

Start of Extract [from: User Guide]

Sorting all contacts : `sort`

Sorts all the contacts in the address book in specified order.
Format: `sort [ORDERING]`

> - Valid orderings are: `name` , `cluster` , `deadline` and `debt` .
> - If no ordering is specified, the address book will be sorted by name in lexicographical order.

Examples:

- `sort`
  Sorts the contacts in the address book by name.
- `sort cluster`
  Sorts the contacts in the address book by their postal districts.

End of Extract

Justification

Debt collectors would want to sort their contacts in various ways for easier viewing.

Implementation

Start of Extract [from: Developer Guide]

Sorting mechanism

Sorting is done within the `UniquePersonList` class.

JAVA

```
public void sortBy(String order) throws IllegalArgumentException {
    switch (order) {
    case "name":
        internalList.sort((Person p1, Person p2) -> p1.getName().compareTo(p2.getName()));
        break;
    case "debt":
        internalList.sort((Person p1, Person p2) -> p2.getDebt().compareTo(p1.getDebt()));
        break;
    case "cluster":
        internalList.sort((Person p1, Person p2) ->
p1.getCluster().compareTo(p2.getCluster()));
        break;
    case "deadline":
        internalList.sort((Person p1, Person p2) ->
p1.getDeadline().compareTo(p2.getDeadline()));
        internalList.sort((Person p1, Person p2) -> Boolean.compare(p1.isWhitelisted(),
p2.isWhitelisted()));
        break;
    default:
        throw new IllegalArgumentException("Invalid sort ordering");
    }
}
```

The `sort` command can take in a `String` that determines how the contacts should be sorted. If no ordering is specified, the contacts will be sorted by ascending lexicographical order by default.

JAVA

```
public class SortCommandParser implements Parser<SortCommand> {
    public SortCommand parse(String args) throws ParseException {
        requireNonNull(args);
        String trimmedArgs = args.trim().toLowerCase();
        switch (trimmedArgs) {
        case "":
        case "name":
        case "debt":
        case "cluster":
        case "deadline":
            return new SortCommand(trimmedArgs);
        default:
            throw new ParseException(String.format(MESSAGE_INVALID_COMMAND_FORMAT,
SortCommand.MESSAGE_USAGE));
        }
    }
}
```

JAVA

```
public static final String DEFAULT_ORDERING = "name";

public SortCommand(String order) {
    //validity of order to sort is checked in {@code SortCommandParser}
    if (order.equals("")) {
        order = DEFAULT_ORDERING;
    }
    this.order = order;
}

public CommandResult execute() throws CommandException {
    try {
        model.sortBy(order);
    } // irrelevant parts of the method omitted for brevity
}
```

Design considerations

Aspect: Default `sort`

Alternative 1 (current choice): `Sort` by `name` by default

Pros: Relatively easy to implement, extremely intuitive.

Alternative 2: `Sort` by `debt` by default

Pros: Equally easy to implement.

Cons: Slightly less intuitive as sorting by `name` is the most prevalent way of sorting contacts.

Alternative 3: No default `sort`

Pros: Extremely easy to implement.

Cons: Not user-friendly.

End of Extract

Enhancement Added: Cluster field

External behaviour

`Clusters` are generated based on based on postal districts

(https://www.ura.gov.sg/realEstateIIWeb/resources/misc/list_of_postal_districts.htm).

Justification

An easy way to group contacts is needed for a debt collector to better plan his/her trips.

Implementation

Start of Extract [from: Developer Guide]

Cluster mechanism

As a debt collector that operates in all parts of Singapore, it would boost efficiency in deciding debt collection trips if the contacts can be effectively grouped by clusters. It is determined based on the postal code provided upon adding a `Person` into the address book. This can be seen in the constructors of the `Person` class and the `Cluster` class.

JAVA
```
public Person(Name name, Phone phone, Email email, Address address, PostalCode postalCode,
              Debt debt, Interest interest, Deadline deadline, Set<Tag> tags) {
    requireAllNonNull(name, phone, email, address, postalCode, debt, interest, deadline, tags);
    // assignment of other fields omitted for brevity
    this.cluster = new SimpleObjectProperty<>(new Cluster(postalCode));
}
```

JAVA
```
public Cluster(PostalCode postalCode) {
    requireNonNull(postalCode);
    if (!isValidPostalCode(postalCode.toString())) {
        throw new AssertionError(MESSAGE_POSTAL_CODE_CONSTRAINTS);
    }
    String cluster = getCluster(postalCode.toString());
    clusterNumber = Integer.parseInt(cluster.substring(0, 2));
    this.value = cluster.substring(4);
}
```

The `getCluster` method resides in the `ClusterUtil` class, and returns the name of the postal district based on the first two numbers of the postal code that is passed into the method. The postal districts are retrieved from [URA (https://www.ura.gov.sg/realEstateIIWeb/resources/misc/list_of_postal_districts.htm)]. The district number is stored as part of the `String` for ease of sorting by location. Part of the code from `ClusterUtil` for retrieving the `cluster` from a postal code starting with `01` is shown below:

JAVA

```
public class ClusterUtil {

    public static final String CLUSTER_POSTAL_DISTRICT_01 = "01. Raffles Place, Cecil, Marina,
People's Park";
    // declaration of other postal districts omitted for brevity
    public static final String CLUSTER_POSTAL_DISTRICT_UNKNOWN = "99. Unknown";

    public static String getCluster(String postalCode) {
        requireNonNull(postalCode);
        int postalSector = Integer.parseInt(postalCode.substring(0, 2));
        switch (postalSector) {
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
        case 6:
            return CLUSTER_POSTAL_DISTRICT_01;
        // cases for all other valid postal districts omitted for brevity
        default:
            return CLUSTER_POSTAL_DISTRICT_UNKNOWN;
        }
    }
}
```

Design Considerations

Aspect: Grouping of contacts

Alternative 1 (current choice): Create a field that contains the name and district number of the postal districts, based on postal code

Pros: Easy to implement and extend, requires minimal user input.

Cons: Requires developer to update `ClusterUtil` whenever a new postal district is drawn by the government, and requires users to reset their contacts' `cluster` via deletion and addition or via editing their `postal codes`.

Alternative 2: Import contacts' location and find their proximity from one another using Google Maps API

Pros: Higher precision of geographical location and proximity.

Cons: Tedious to implement proximity. Either takes up too much time in generating nearby contacts or too much space in storing them.

Alternative 3: Assign general location to each contact upon addition into the address book or via editing

Pros: Easy to implement.

Cons: Tedious for users. They also have to ensure that they do not make spelling mistakes.

End of Extract

Enhancement Added: Nearby command

External behavior

Start of Extract [from: User Guide]

Selecting a nearby person: `nearby`

Selects the person identified by the index number used in the listing of nearby contacts of currently selected person,
Format: `nearby INDEX`

- A person must be selected before this command is called.
- Selects the person and loads the full information of the person at the specified `INDEX`.
- The index refers to the index number shown in the nearby contacts listing.
- The index **must be a positive integer** (e.g. 1, 2, 3, …)

Examples:

- ```
  list
  select 2
  nearby 1
  ```
  Selects the 1st person in the same `cluster` as the previously selected person.

End of Extract

Justification

Debt collectors may want to see details of nearby contacts to plan a more effective visiting schedule.
  ```

Enhancement Added: Making index optional for commands

External behavior

Start of Extract [from: User Guide] written by Jelena Neo Hui Ling

Increasing the debt of a debtor: `borrow`

Increases the debt of a debtor by the amount entered.
Format: `borrow [INDEX] AMOUNT`

- Increases the debt and total debt of the debtor at the specified `INDEX` by `AMOUNT`. The index refers to the index number shown in the last person listing. The index **must be a positive integer** (e.g. 1, 2, 3, …)
- If no index is specified, the debt of the currently selected person is updated instead.
- `AMOUNT` has to be in dollars and cents. For example: `500.50` which represents $500.50.
- This command also sets the date repaid to `NOT REPAID` if the person previously fully repaid his/her debts.

Examples:

- `borrow 1 500`
  Increases the debt of the 1st person by $500.
- `borrow 2 1000.10`
  Increases the debt of the 2nd person by $1000.10.
- `list`
  `select 2`
  `borrow 234`
  Increases the debt of the 2nd person by $234.

End of Extract

Justification

Calling commands directly on the currently selected person makes much more intuitive sense than always supplying an `INDEX` each time. This also increases the flexibility and convenience of Codii.

Implementation

Start of Extract [from: Developer Guide]

Optional command indexes

It is intuitive to allow commands such as `edit`, `delete`, `borrow` and others to be called on the currently selected person instead of always having to supply the `INDEX`.
Parsers of commands call a constructor of the commands without an index. Take the `RepaidCommandParser` and `RepaidCommand` for example.

JAVA

```
public RepaidCommand parse(String args) throws ParseException {
    try {
        if (args.trim().equals("")) {
            return new RepaidCommand();
        } else {
            Index index = ParserUtil.parseIndex(args);
            return new RepaidCommand(index);
        }
    } catch (IllegalValueException ive) {
        throw new ParseException(
            String.format(MESSAGE_INVALID_COMMAND_FORMAT, RepaidCommand.MESSAGE_USAGE));
    }
}
```

JAVA

```
public RepaidCommand() throws CommandException {
    personToWhitelist = selectPersonForCommand();
}

public RepaidCommand(Index targetIndex) throws CommandException {
    personToWhitelist = selectPersonForCommand(targetIndex);
}
```

The `selectPersonForCommand()` and `selectPersonForCommand(Index)` methods are placed in the `Command` class, and is used by such index-based commands to select the currently selected person to apply the command on if no index is provided.

JAVA

```
public ReadOnlyPerson selectPersonForCommand() throws CommandException {
    if (ListObserver.getSelectedPerson() == null) {
        throw new CommandException(Messages.MESSAGE_NO_PERSON_SELECTED);
    }
    return ListObserver.getSelectedPerson();
}

public ReadOnlyPerson selectPersonForCommand(Index index) throws CommandException {
    List<ReadOnlyPerson> lastShownList = ListObserver.getCurrentFilteredList();
    if (index.getZeroBased() >= lastShownList.size()) {
        throw new CommandException(Messages.MESSAGE_INVALID_PERSON_DISPLAYED_INDEX);
    }
    return lastShownList.get(index.getZeroBased());
}
```

Design Considerations

Aspect: Executing index-based commands without index

Alternative 1 (current choice): The person to apply the commands on are determined in the constructors

Pros: Allows for proper `redo`.

Cons: Hard to test as Command Exceptions are being thrown from the constructors.

Alternative 2: Calling a constructor of the `command` without an `Index` initialises its `targetIndex` to null, and the null index is handled as a special value during execution

Pros: Easy to implement, easy to extend.

Cons: Although highly unlikely, it may be possible for a `command` to have an unintended null `targetIndex`, which will then cause it to behave as an indexless command instead of an error. May also cause problems with `redo`.

End of Extract

Enhancement Added: Switching themes

External behavior

Start of Extract [from: User Guide]

Changing themes: `theme`

Changes between the two available themes shown below in Figures 4.25.1 and 4.25.2 below.

Format: `theme`

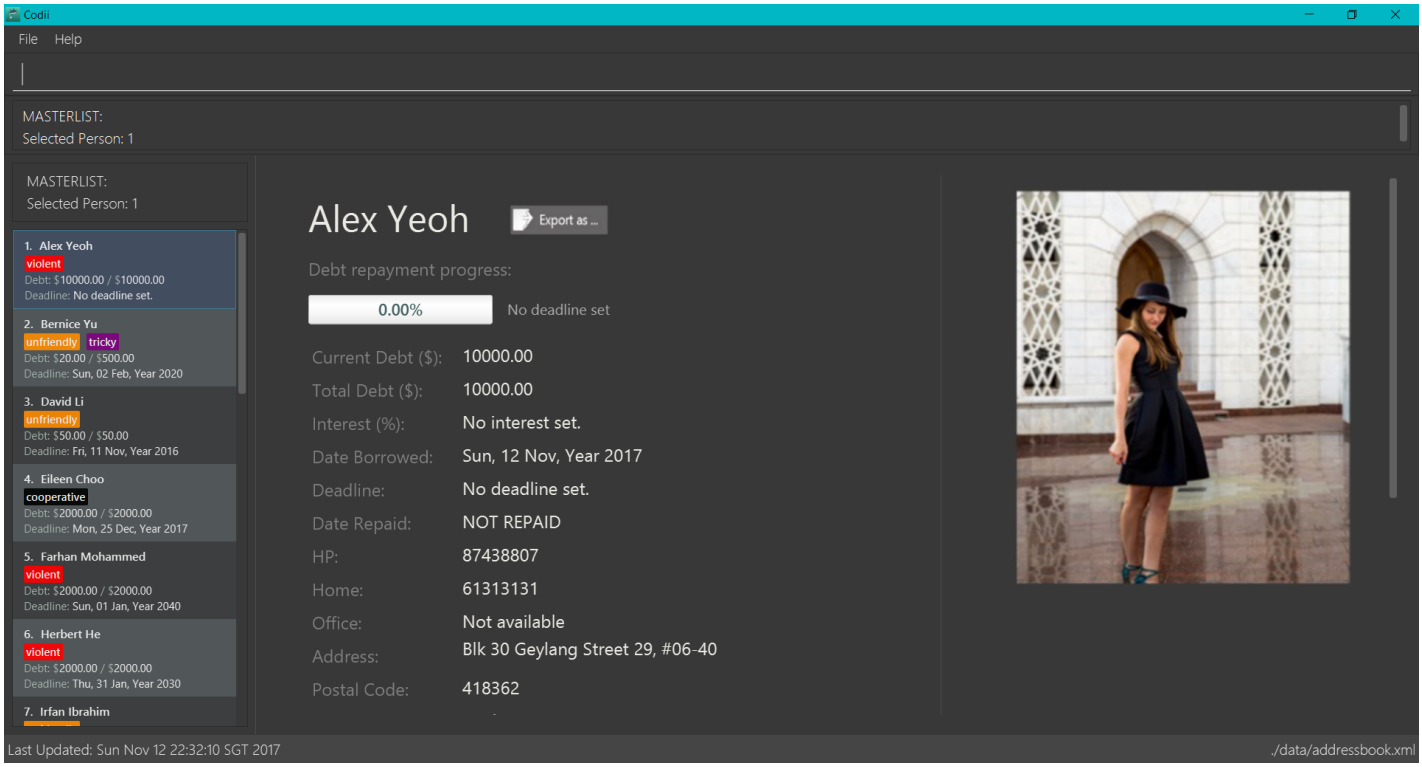

Figure 4.25.1 : Dark theme (default)

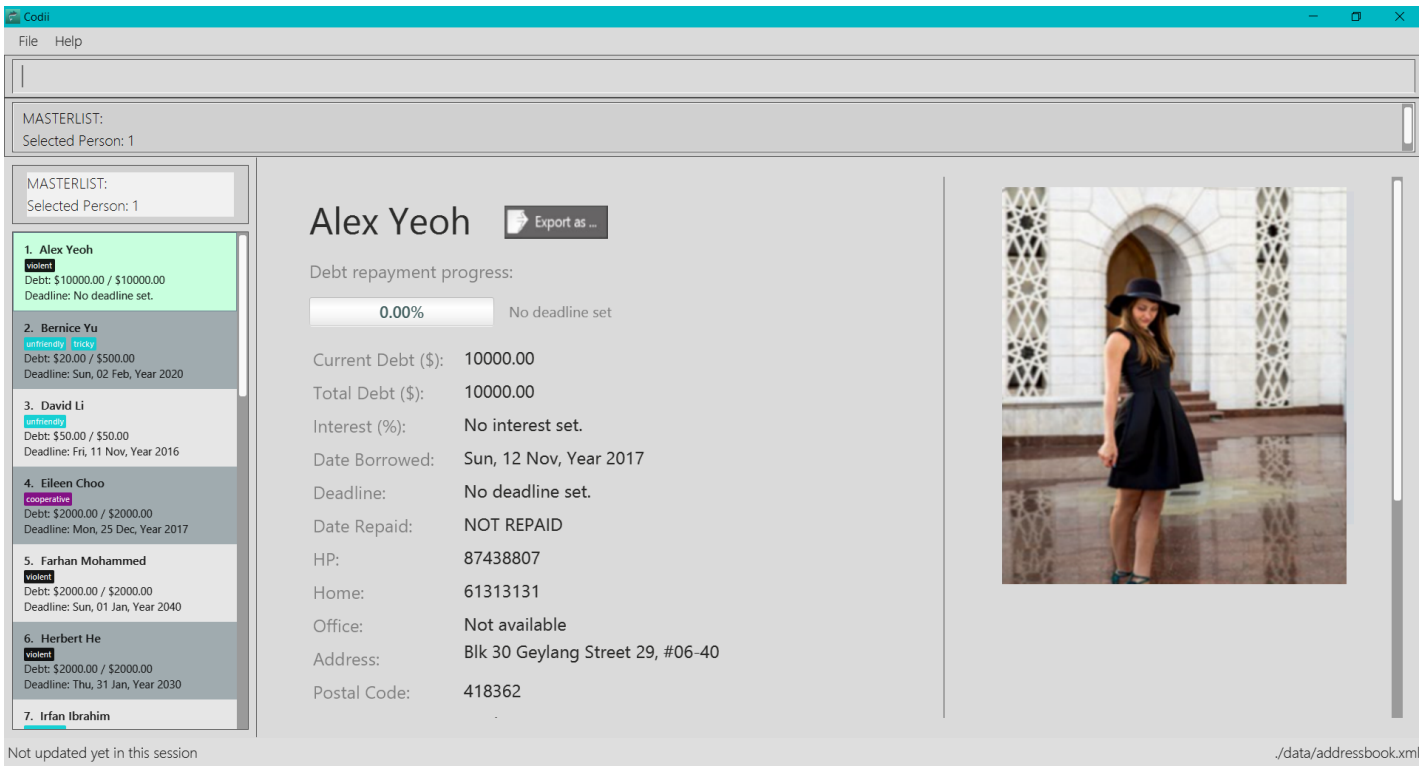

Figure 4.25.2 : Bright theme

End of Extract

Justification

The ability to change themes can make the user feel more as though the app belongs to him/her as it is a form of customisation. It allows for a much better user experience.

Implementation

Start of Extract [from: Developer Guide]

Theme Changing Mechanism

The changing of themes is done in the `MainWindow` class, which is the class that holds all the `UI` parts.

JAVA

```
private void changeTheme() {
    for (String stylesheet : getRoot().getStylesheets()) {
        if (stylesheet.endsWith("DarkTheme.css")) {
            getRoot().getStylesheets().remove(stylesheet);
            getRoot().getStylesheets().add("/view/BrightTheme.css");
            break;
        } else if (stylesheet.endsWith("BrightTheme.css")) {
            getRoot().getStylesheets().remove(stylesheet);
            getRoot().getStylesheets().add("/view/DarkTheme.css");
            break;
        }
    }
}
```

The `changeTheme` method is called when a `ChangeThemeRequestEvent` is raised from `ThemeCommand`.

In `MainWindow`:

```java
@Subscribe
private void handleChangeThemeRequestEvent(ChangeThemeRequestEvent event) {
    logger.info(LogsCenter.getEventHandlingLogMessage(event));
    changeTheme();
}
```

In `ThemeCommand`:

```java
public CommandResult execute() {
    EventsCenter.getInstance().post(new ChangeThemeRequestEvent());
    return new CommandResult(MESSAGE_SUCCESS);
}
```

Design Considerations

Aspect: Condition for changing themes

Alternative 1 (current choice): Use `String#endsWith()`

Pros: Foolproof, guaranteed to work if the theme exists.

Alternative 2: Use file paths

Pros: Likely to work most of the time if handled well.

Cons: When debugging, it was found that the path started in the `build` folder instead of the `src` folder. Using this method seemed to cause inconsistencies between running the app from an IDE and from the `.jar` file.

Aspect: Switching of themes

Alternative 1 (current choice): Simple toggle between two themes

Pros: Easy to implement.

Cons: Troublesome to extend.

Alternative 2: Use an `int` to keep track of the current theme, assign each theme to a number, and use `+` and `%` to cycle through the themes

Pros: Easy to extend.

Cons: Troublesome to implement considering that we intend to use only two themes.

End of Extract

Other contributions

Various UI enhancements

- Replacing the browser with full information panel (Pull request #48 (https://github.com/CS2103AUG2017-T17-B1/main/pull/48))

- Creating the nearby persons panel inside the full information panel (Pull request #128 (https://github.com/CS2103AUG2017-T17-B1/main/pull/128))
- Adding a display for current list and selected index (Pull request #249 (https://github.com/CS2103AUG2017-T17-B1/main/pull/249))

Miscellaneous

- Restructure commands (Pull request #296 (https://github.com/CS2103AUG2017-T17-B1/main/pull/29))
- Raise and fix bugs (https://github.com/CS2103AUG2017-T17-B1/main/issues?utf8=%E2%9C%93&q=is%3Aissue%20label%3Abug%20)
- Assist teammates with debugging tests
- Report bugs and raise suggestions for another team (Issues #78 (https://github.com/CS2103AUG2017-T10-B3/main/issues/78), #76 (https://github.com/CS2103AUG2017-T10-B3/main/issues/76), #82 (https://github.com/CS2103AUG2017-T10-B3/main/issues/82))
- Offer a bug fix for reuse (Issue #196 (https://github.com/nus-cs2103-AY1718S1/forum/issues/196))
- Help others on Slack
    - Suggest alternative for importing Address Book level 4 for those facing issues with gradle
    - Remind others to delete .json files and the data/ folder if they were doing acceptance testing for another team in the same folder that they saved their own .jar file.

Project: Dog Mario

Dog Mario (https://github.com/DogMario/Dog-Mario) is a PC game developed with Tan Bing Hwang (https://github.com/tbhbhbh) as an independent project for CP2106 Independent Software Development Project (Orbital) (http://nusskylab-dev.comp.nus.edu.sg/).

Last updated 2017-11-13 23:41:44 +08:00